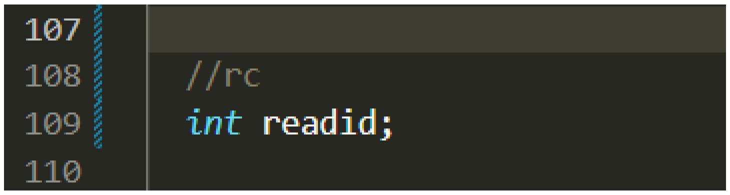
Mini Project 2: Enhancing XV6

SPECIFICATION 1: SYSTEM CALLS

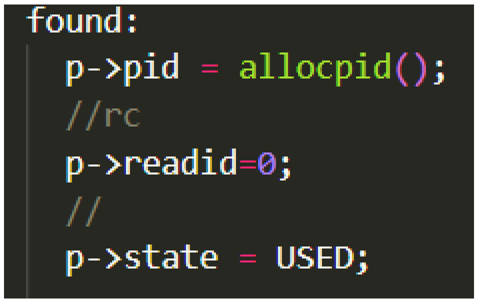
System call 1: getreadcount()

- The system call returns the value of a counter which is incremented every time any process calls the *read()* system call.
- We need to store the count on a process basis so we will have to add something to the structure created for a process.
- In `proc.h`, the structure for a process is defined. Find struct proc and then add an integer to keep account for the read calls. eg. `int readid;`



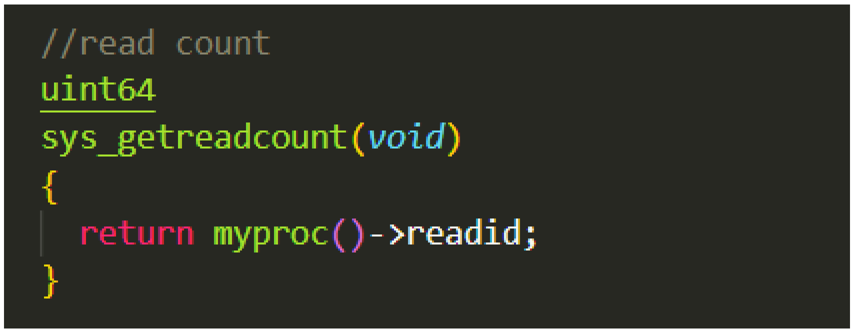
```

//rc
int readid;

```

- In `proc.c` the function allocproc allocates processes and it can be seen that it jumps to label found where it is assigned the pid. Added a line here to initialize the readcount to zero in this case `p->readid = 0;`



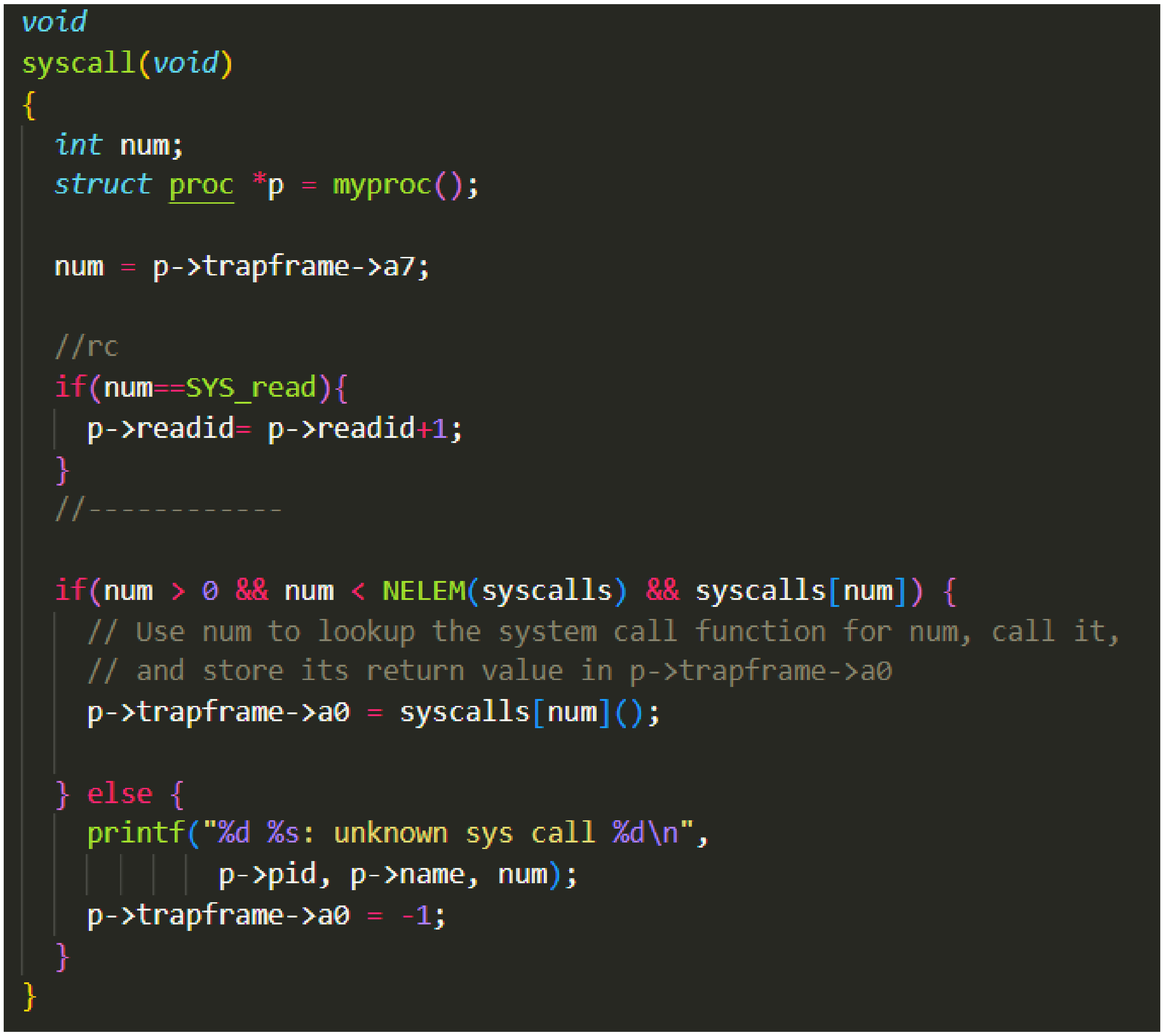
```
found:
  p->pid = allocpid();
  //rc
  p->readid=0;
  //
  p->state = USED;
```

- The functions in `sysproc.c` can access the process structure of a given process by calling `myproc()`. Using this, we define the function sys_getreadcount which takes no arguments and returns an int.



```c
//read count
uint64
sys_getreadcount(void)
{
  return myproc()->readid;
}
```

- In `syscall.c` , i made the following change:



```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;

  //rc
  if(num==SYS_read){
    p->readid= p->readid+1;
  }
  //------------

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();

  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

- every system call is mapped to something in syscall.c. In `syscall.h` i added an entry `#define SYS_getreadcount 27`

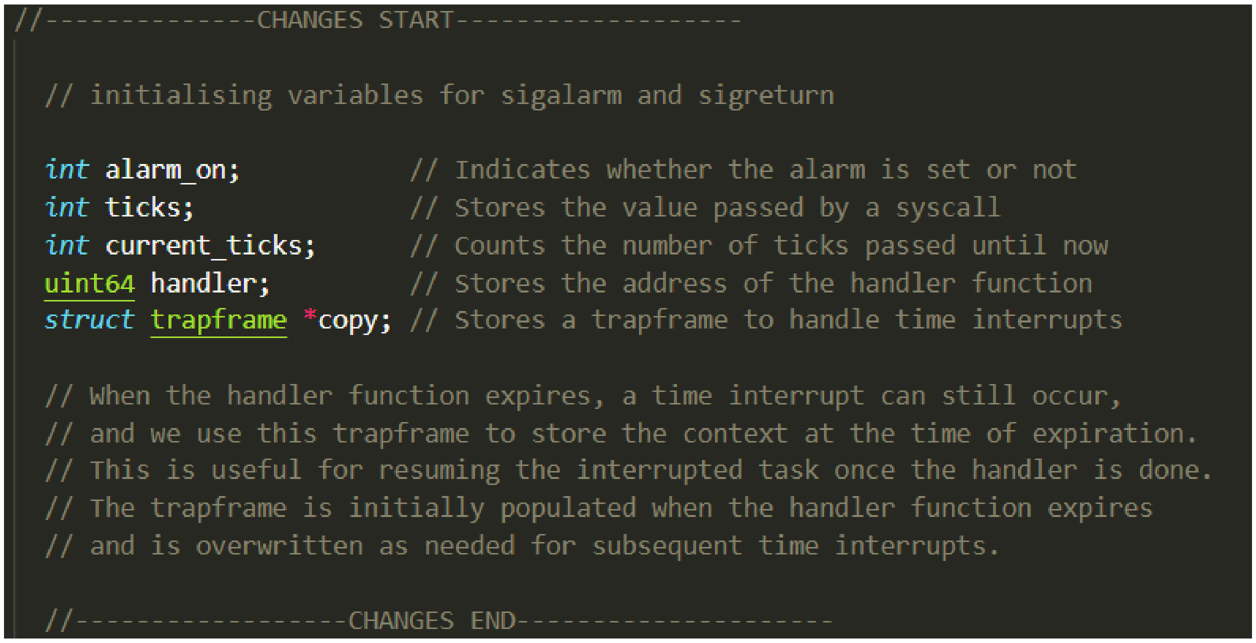
- Now in `syscall.c` , I added the entry `[SYS_getreadcount] sys_getreadcount, //my change` to define that it maps to `sys_getreadcount`.
- As `getreadcount` is also an external function I added a line to tell that to the compiler `extern int sys_getreadcount(void);`

System call 2: sigalarm and sigreturn

- Added the system calls sigalarm and sigreturn and a user program alarmtest. The command will be executes as followed:

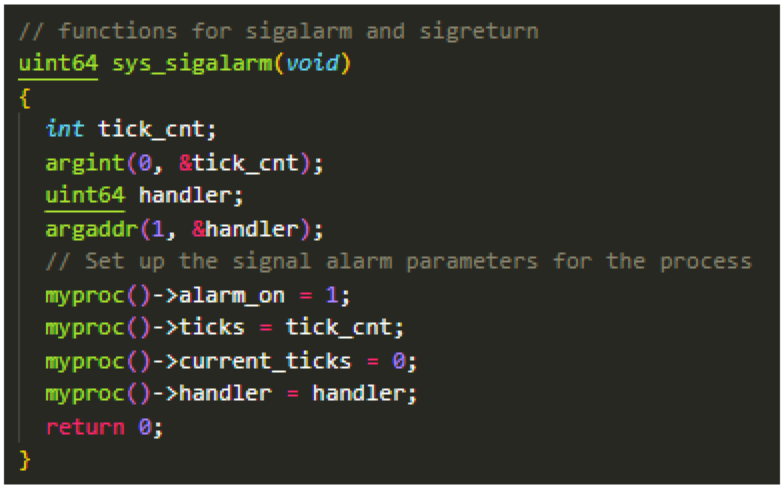
```
$ alarmtest
```

- Add two syscalls sigalarm and sigreturn in `user/user.h`
- Add `$U/_alarmtest\` in UPROGS in makefile
- Add the mentioned variables in `kernel/proc.h` in struct proc



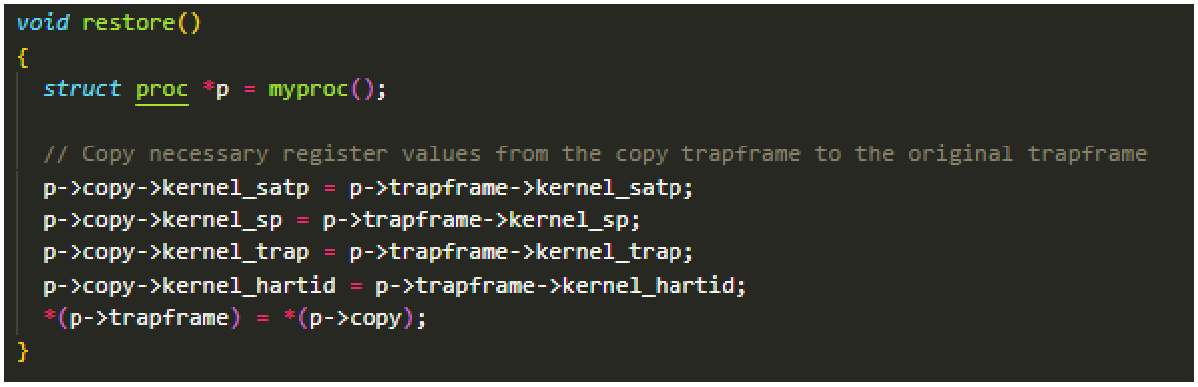
```
//--------------CHANGES START-------------------

  // initialising variables for sigalarm and sigreturn

  int alarm_on;            // Indicates whether the alarm is set or not
  int ticks;               // Stores the value passed by a syscall
  int current_ticks;       // Counts the number of ticks passed until now
  uint64 handler;          // Stores the address of the handler function
  struct trapframe *copy; // Stores a trapframe to handle time interrupts

  // When the handler function expires, a time interrupt can still occur,
  // and we use this trapframe to store the context at the time of expiration.
  // This is useful for resuming the interrupted task once the handler is done.
  // The trapframe is initially populated when the handler function expires
  // and is overwritten as needed for subsequent time interrupts.

  //------------------CHANGES END---------------------
```

- Implemented sys_sigalarm , sys_sigreturn and a restore function in `kernel/sysproc.c`
  - sys_sigalarm



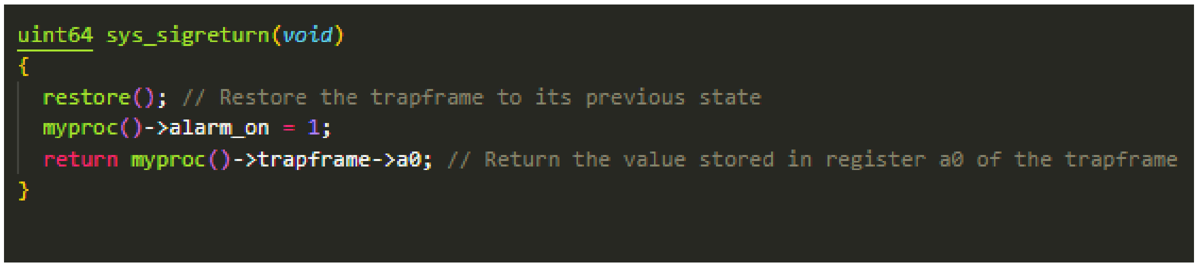
```c
// functions for sigalarm and sigreturn
uint64 sys_sigalarm(void)
{
  int tick_cnt;
  argint(0, &tick_cnt);
  uint64 handler;
  argaddr(1, &handler);
  // Set up the signal alarm parameters for the process
  myproc()->alarm_on = 1;
  myproc()->ticks = tick_cnt;
  myproc()->current_ticks = 0;
  myproc()->handler = handler;
  return 0;
}
```

- restore



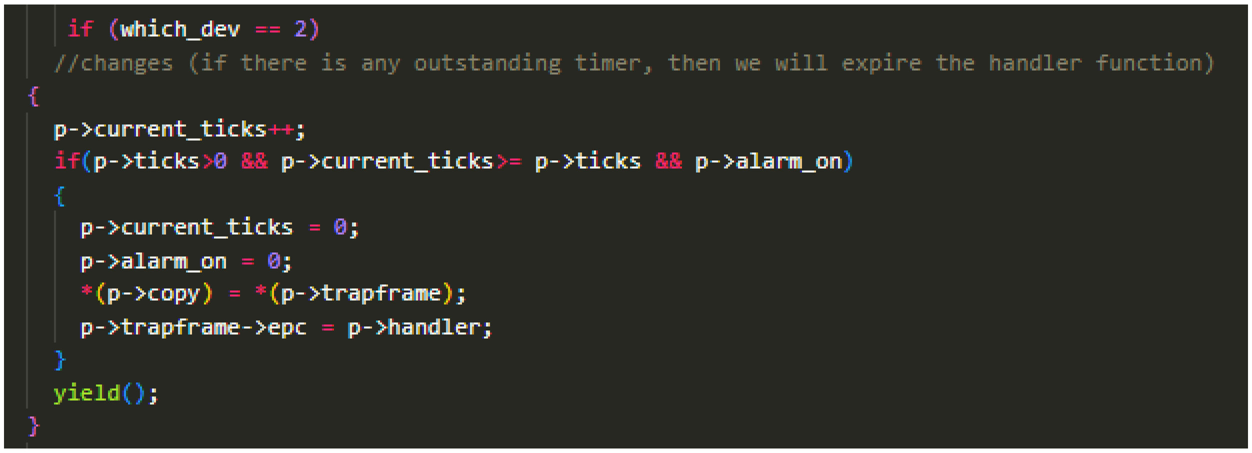
```c
void restore()
{
  struct proc *p = myproc();

  // Copy necessary register values from the copy trapframe to the original trapframe
  p->copy->kernel_satp = p->trapframe->kernel_satp;
  p->copy->kernel_sp = p->trapframe->kernel_sp;
  p->copy->kernel_trap = p->trapframe->kernel_trap;
  p->copy->kernel_hartid = p->trapframe->kernel_hartid;
  *(p->trapframe) = *(p->copy);
}
```

- sys_sigreturn



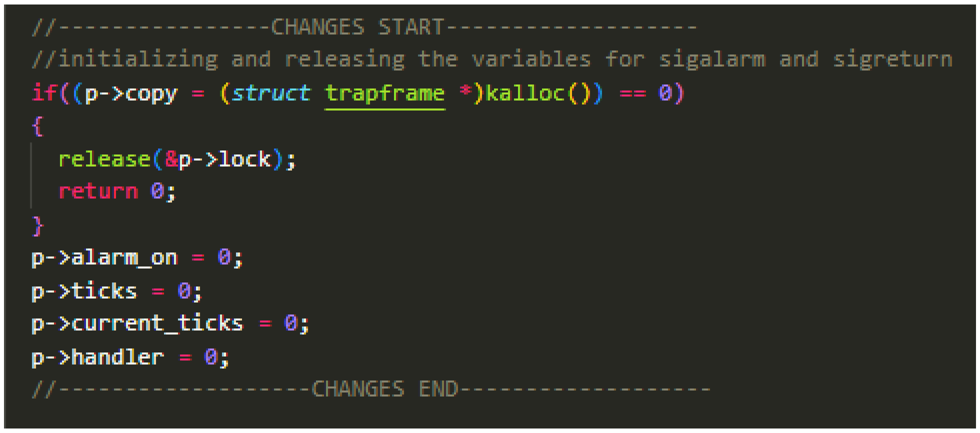
```c
uint64 sys_sigreturn(void)
{
  restore(); // Restore the trapframe to its previous state
  myproc()->alarm_on = 1;
  return myproc()->trapframe->a0; // Return the value stored in register a0 of the trapframe
}
```

- Made some changes in `kernel/trap.c` program to handle when the ticks will reach maximum and will be then handled by epc.



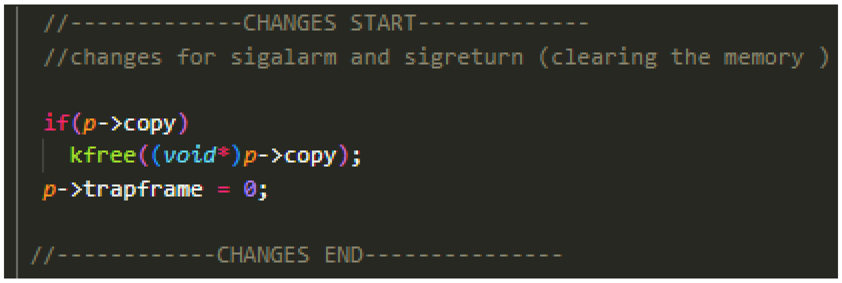
```c
    if (which_dev == 2)
  //changes (if there is any outstanding timer, then we will expire the handler function)
{
  p->current_ticks++;
  if(p->ticks>0 && p->current_ticks>= p->ticks && p->alarm_on)
  {
    p->current_ticks = 0;
    p->alarm_on = 0;
    *(p->copy) = *(p->trapframe);
    p->trapframe->epc = p->handler;
  }
  yield();
}
```

- Initialized new variables and withdrew them in `kernel/proc.c`
  - allocproc



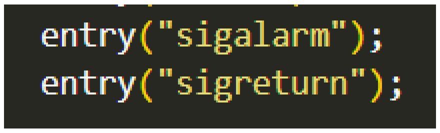
```c
//----------------CHANGES START-------------------
//initializing and releasing the variables for sigalarm and sigreturn
if((p->copy = (struct trapframe *)kalloc()) == 0)
{
  release(&p->lock);
  return 0;
}
p->alarm_on = 0;
p->ticks = 0;
p->current_ticks = 0;
p->handler = 0;
//------------------CHANGES END-------------------
```

  - freeproc



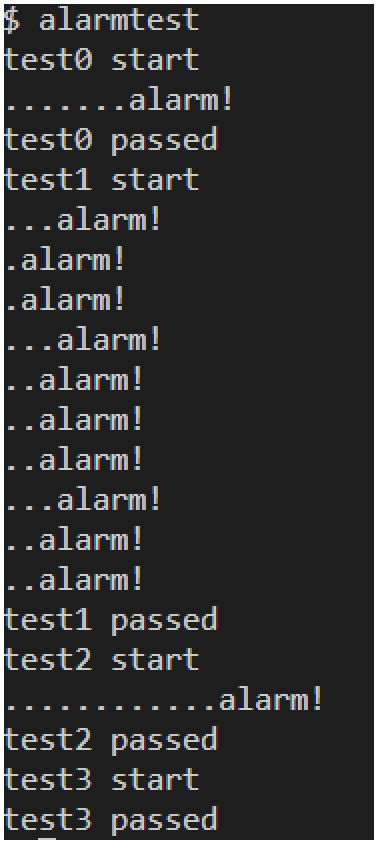
```c
  //-------------CHANGES START-------------
  //changes for sigalarm and sigreturn (clearing the memory )

  if(p->copy)
    kfree((void*)p->copy);
  p->trapframe = 0;

 //------------CHANGES END---------------
```

- Added the entries in user/usys.pl



```
entry("sigalarm");
entry("sigreturn");
```

- Run command `alarmtest` on shell and this will be the output.



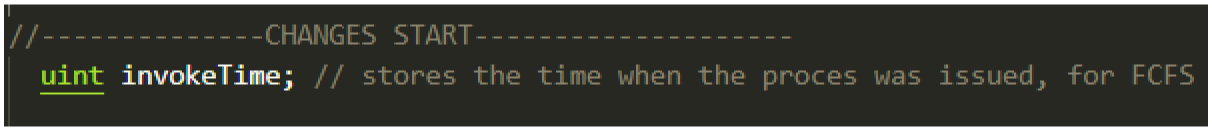
```
$ alarmtest
test0 start
.......alarm!
test0 passed
test1 start
...alarm!
.alarm!
.alarm!
...alarm!
..alarm!
..alarm!
..alarm!
...alarm!
..alarm!
..alarm!
test1 passed
test2 start
............alarm!
test2 passed
test3 start
test3 passed
```

SPECIFICATION 2: SCHEDULING

FCFS

- Implement a policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created). The process will run until it no longer needs CPU time.
- FCFS is a non-preemptive scheduler that schedules the program that was invoked the earliest.

- This is done by finding the program with the smallest invoking time (in ticks) and then setting it to run.
- It runs until it completes, the next process is found, and so on.
  Added
  invoke time variable in struct proc in `kernel/proc.h`



```
//--------------CHANGES START--------------------
  uint invokeTime; // stores the time when the proces was issued, for FCFS
```

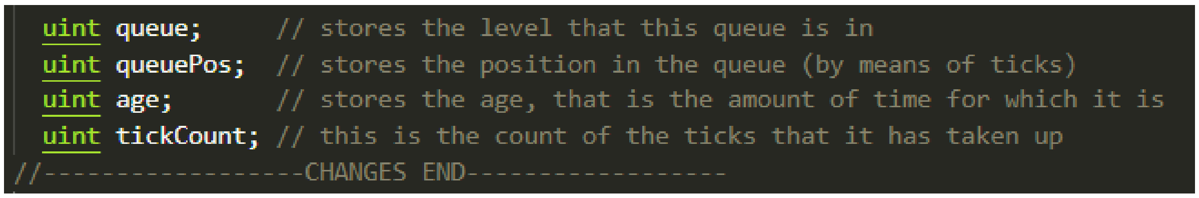
- Defined a program for FCFS in the scheduler function in `kernel/proc.c`

```
#ifdef FCFS

    struct proc *currentProc = NULL; // this is the optimal proc
    uint earliest = 4294967295;      // setting to maximum possi
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);
      if (p->invokeTime < earliest)
      {
        if (p->state == RUNNABLE)
        {
          currentProc = p;
          earliest = p->invokeTime;
        }
      }
      release(&p->lock);
    }

    if (currentProc != NULL)
    {
      acquire(&currentProc->lock);
      if (currentProc->state == RUNNABLE)
      {

        currentProc->state = RUNNING;
        c->proc = currentProc;
        swtch(&c->context, &currentProc->context);

        c->proc = 0;
      }
      release(&currentProc->lock);
    }

#endif
```

MLFQ- Multi level feedback queue

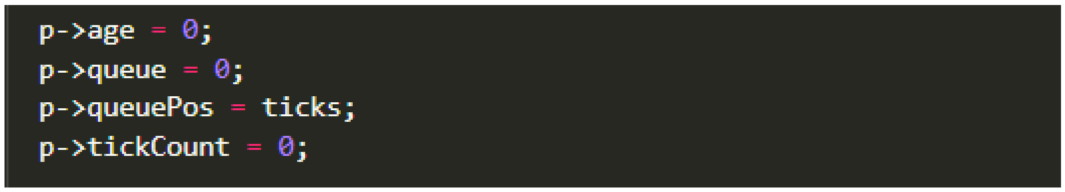
- This makes 4 queues represent decreasing priorities. All processes that enter are assigned a priority queue 0, which is the highest priority.
- Each of the queues have associated with it a time slice so that a process in that queue gets to run for that time slice before it is preempted.
- Instead of making real queues, we emulate the behaviour of a queue by adding fields in the process to represent the queue level, and a field that represents the time it was inserted into the "queue", which helps us keep track of which processes in a given queue was inserted the earliest. Processes inserted earlier have a higher priority in the queue.
- `Trap.c` increments the running time (in ticks) of a running process, and waiting time (measured by age) for runnable process waiting in the queue.
- If a process crosses the assigned time slice (for its level of a queue, then it is preempted by pushing it to the end of the next queue (by incrementing its queue field and reseting the queue position to the current ticks)

- `Trap.c` also keeps incrementing the age of a process in the scheduler, and once it crosses a certain limit (we chose 32), we push it to the back end of a queue above it to raise the priority. By doing so, we prevent starvation.
- `Trap.c` does once final thing: on each clock tick it checks our entire multi level queue for processes that have a queue greater than the current queue. If it finds something, it preempts the current running process and pushes it to the back of the current(by resetting ticks). Then it calls yield which re runs the scheduler which will find the process with the higher priority and run that instead.
- In `proc.c`, apart from initializing the fields, we simply enumerate all process into their respective queues. (By making a temporary multilevel array). Then we iterate step by step through each queue, find the process with the lowest ticks (since that is the start of the queue) and run it.
- The process that is running has to be locked. (This continues in a for loop until all process have been run, or trap.c has requested rescheduling.)
- Added fields in struct proc in `kernel/proc.h` for queue position and initialised them in `kernel/proc.c`
    - `kernel/proc.h`



```
  uint queue;     // stores the level that this queue is in
  uint queuePos;  // stores the position in the queue (by means of ticks)
  uint age;       // stores the age, that is the amount of time for which it is
  uint tickCount; // this is the count of the ticks that it has taken up
//------------------CHANGES END------------------
```

    - `kernel/proc.c`



```
  p->age = 0;
  p->queue = 0;
  p->queuePos = ticks;
  p->tickCount = 0;
```

- Implemented MLFQ logic in kernel/trap.c

```
#ifdef MLFQ
  int t_slice[4] = {1,3,9,15};
  if (which_dev == 2)
  {
    struct proc *p;
     // Iterate through all processes
    for (p = proc; p < &proc[NPROC]; p++)
    {
      // acquire(&p->lock);
      // we dont need to acquire a lock since this will be teste

      if (p->state == RUNNABLE)
      {
        p->age++;
        if (p->age > 32)
        { // choosing 32 as the limit of ageing before starvatio
          if (p->queue > 0)
          { // ageing does not apply to to priority 0
            p->queue--;
            p->age = 0;       // Reset waiting time
            p->tickCount = ticks; // pushing it to the back of t
          }
        }
      }
    }
    // Get the currently running process
    struct proc *currentProc = myproc();
   // Check if there is a process with higher priority running.
  //  if there is then preempt curr process.

    if (currentProc && currentProc->state == RUNNING)
    {


        currentProc->tickCount++;
      // checking for higher priority
      for (p = proc; p < &proc[NPROC]; p++)
      {
        if (p->queue < currentProc->queue)
        {
          currentProc->queuePos = ticks;
       // pushing back to the end of the current queue
          // printf("priority preemption\n");
          yield();// Preempt the current process
        }
      }

    // Check if the current process has exhausted its time slice
      if (currentProc->tickCount >= t_slice[currentProc->queue]
      {
        // preempt the process by moving the process to the next
        // printf("here");
        currentProc->tickCount = 0;
        if (currentProc->queue < 4)
        {
          currentProc->queue++;
        }
        currentProc->tickCount = 0;
        p->queuePos = ticks; // we use ticks to keep track of th
        yield();
      }
    }
  }
#endif
```

- Implementing scheduler logic in kernel/proc.c

```
#ifdef MLFQ
    // we make 4 arrays, representing all 4 queues, and then fir
    struct proc *procArray[4][NPROC];
    for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < NPROC; j++)
      {
        procArray[i][j] = 0;
      }
    }
    int idxArray[4] = {0};
    for (p = proc; p < &proc[NPROC]; p++)
    {

      if (p->state == RUNNABLE)
      {

        procArray[p->queue][idxArray[p->queue]++] = p;
      }
    }

    // now, find the array with the smallest size, find the arra

    for (int i = 0; i < 4; i++)
    {

      while (idxArray[i]--)
      {

        struct proc *bestProc = 0; // this is the proc with the
        struct proc *iteratorProc;
        uint lowestTicks = 4294967295;
        int procPos = 0;

        for (int j = 0; j < NPROC; j++)
        {

          if (procArray[i][j])
          {
            if (procArray[i][j]->tickCount < lowestTicks)
            {
              lowestTicks = iteratorProc->tickCount;
              bestProc = procArray[i][j];
              procPos = j;
            }
          }
        }

        // we now set the best process to run, after resetting t
        lowestTicks = 4294967295;
        if (bestProc && bestProc->state == RUNNABLE)
        {
          procArray[i][procPos] = 0;
          bestProc->state = RUNNING;
          bestProc->age = 0;
          acquire(&bestProc->lock);
          c->proc = bestProc;
          swtch(&c->context, &bestProc->context);
          c->proc = 0;
          release(&bestProc->lock);
        }
        bestProc = 0;

        // printf("\n\n\n HERE %d\n\n\n", tester++);
      }
    }
#endif
```

COMPARISON BETWEEN THE SCHEDULERS:

Scheduling Type	Waiting time	Running Time
FCFS	144	15
MLFQ	138	12

SPECIFICATION 4: NETWORKING

PART B:

1. How is **your** implementation of data sequencing and retransmission different from traditional TCP? (If there are no apparent differences, you may mention that)

    - The implementation of data sequencing and retransmission in our UDP-based solution differs significantly from traditional TCP in several ways:
    - **Reliability:** TCP is a reliable, connection-oriented protocol that guarantees the delivery of data packets in the correct order. Our UDP-based solution is connectionless and does not guarantee the same level of reliability. While we implement retransmissions for lost packets, it's important to note that UDP does not provide the same built-in reliability mechanisms as TCP.
    - **Acknowledgments:** In TCP, acknowledgments (ACKs) are used to confirm the receipt of data packets. In our UDP implementation, we simulate the sending and receiving of ACKs for data chunks to demonstrate retransmissions. However, this simulation is not a standard UDP behavior and is for illustrative purposes only. TCP handles acknowledgments more robustly with a sliding window mechanism.
    - **Flow Control:** TCP includes a flow control mechanism to prevent sender overrun receiver's buffer. Our UDP implementation does not include flow control, and it relies on the sender to manage the pace of data transmission. UDP does not inherently support flow control mechanisms like TCP's window-based approach.
    - **Ordering:** TCP guarantees in-order delivery of data packets, which means that if data packets arrive out of order, they will be reordered before delivery. In UDP, the order of packet arrival is not guaranteed, and it's up to the application layer to handle reordering if needed.

2. How can you extend **your** implementation to account for flow control? You may ignore deadlocks.

    - To extend our current implementation to account for flow control, we can implement a flow control mechanism similar to TCP's sliding window approach. Here are the key steps to achieve this:

- **Sender's Side:**
    - Maintain a sender's window size, indicating the maximum number of unacknowledged data chunks that can be in transit at any given time.
    - Keep track of the receiver's advertised window size, indicating the amount of buffer space available at the receiver.
    - Before sending a new data chunk, check if the number of unacknowledged chunks is less than the sender's window size and if it fits within the receiver's advertised window.
    - If the conditions are met, send the data chunk. Otherwise, wait until more space becomes available.
- **Receiver's Side:**
    - Maintain a receiver's window size, indicating the maximum number of out-of-order data chunks it can accept.
    - Send ACKs for received data chunks to inform the sender of the available buffer space at the receiver.
    - When a data chunk is received out of order, buffer it until the missing chunks are received and can be processed in order.
- **Adjustments:**
    - Dynamic adjustment of the window sizes based on network conditions and available resources can be implemented to optimize flow control.
- **Handling Retransmissions:**
    - Continue with the retransmission mechanism as previously implemented for lost packets.
- **Deadlock Consideration:**
    - While not explicitly required, deadlock avoidance should be taken into account by implementing timeout mechanisms for the sender to resend packets if acknowledgments are not received within a reasonable time.

Incorporating these modifications would make the UDP-based implementation more TCP-like in terms of flow control while maintaining the core benefits of UDP, such as low overhead and minimal connection setup.